ACLMPL: Portable and Efficient Message Passing for MPPs

James Painter, Patrick McCormick*,
Michael Krogh, Charles Hansen
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Guillaume Colin de Verdière†
Centre d'Etudes de Limeil-Valenton
CEL-V/DMA/AIM
94195 Villeneuve-Saint-Georges, France

September 19, 1995

Abstract

This paper presents the Advanced Computing Lab Message Passing Library (ACLMPL). Modeled after Thinking Machines Corporation's CMMD, ACLMPL is a high throughput, low latency communications library for building message passing applications. The library has been implemented on the Cray T3D, Thinking Machines CM-5, SGI workstations, and on top of PVM. On the Cray T3D, benchmarks show ACLMPL to be 4 to 7 times faster than MPI or PVM.

1 Introduction

Parallel programs are typically written in one of two styles: SIMD or MIMD. SIMD (Single Instruction, Multiple Data) programs are typically written in a data parallel language, such as Fortran 90[1]. With such programs interprocessor communication is hidden from the programmer via the compiler and runtime system.

With MIMD (Multiple Instruction, Multiple Data) programs the developer typically calls message passing primitives to perform interprocess communications. For such programs to run efficiently and gain the best possible speedup, the communications cost of the program must be as *small* as possible. By small, we

*Author additionally with the Computer Science Department at The University of New Mexico
†Author currently at the ACL through a grant from DGA/DRET
[1]SPMD (Single Program, Multiple Data) can be thought of as a more general form of SIMD.

mean lowest latency and highest throughput. If communications costs are high, then the program will be severely limited in terms of its speedup potential as the number of processors increases.

The development of the ACL Message Passing Library (ACLMPL) was driven by two motivations: performance and portability.

As already mentioned, performance of a communications library is crucial to overall program performance. This fact was made all too clear when we started porting our message passing programs from the Thinking Machines Corporation (TMC) CM-5 massively parallel computer to the Cray Research Inc. (CRI) T3D. CRI supplies an implementation of the Parallel Virtual Machine (PVM) message passing library for the T3D; however, in our experience, its performance is poor. We found that our codes not only performed poorly but that they did not scale up (or run in some cases) to large numbers of processors. This was entirely due to the implementation of PVM on the T3D.

Our second motivation, portability, was driven by our investment in CM-5 software development. Many of the libraries and programs that we have developed for the CM-5 use a message passing library called CMMD. Upon the arrival of our T3D, we wanted an easy migration path for our software. Additionally, the CM-5 will still be a major production machine for some time to come. Thus having a common messaging system would be an added bonus.

The remaining sections describe previous message passing systems, describe the implementation of ACLMPL, present timings, describe a few applications that use ACLMPL, and draw conclusions.

2 Previous Work

The PVM library was designed to treat a collection of computers, which may be workstations, servers, vector computers, or even MPPs, as a single distributed parallel computer [4]. To accomplish this, PVM supports heterogeneous processors, networks and data types. Besides basic communication primitives (asynchronous send and receive), PVM has primitives for process control, synchronization, signaling, process groups, and virtual machine control.

The Message Passing Interface (MPI) library was designed with efficiency and portability in mind. The MPI feature set was designed by a committee which used features and concepts from many various message passing systems [5]. What resulted is a "full-featured" message passing library that includes many variations on send and receive (blocking/nonblocking, buffered/unbuffered, receiver-ready, different data types including user specified, and more). Additionally, MPI includes support for global operations (barriers, reductions, gather/scatters, broadcasts, scans, etc.), processor topologies, processor groups, profiling, and error handling. Process management (creation, deletion, migration), active messages, and I/O support are not included in the current standard but are expected to be in the MPI-2 standard.

The Illinois Fast Messaging library (FM) has the goal of providing a fast message passing interface [7]. The authors of FM tailor the internals of the library to the particular architecture that it is being used on. FM

provides a standard interface and high performance; however, it forces the user into using active messages instead of synchronous or asynchronous communications primitives. Also, FM does not provide global operations. Additionally, it has a fairly small message size constraint and in its current implementation it has compatibility problems with PVM, MPI, and shared memory operations on the T3D.

TMC created CMMD for the CM-5 massively parallel computer [3]. CMMD supports three styles of communication: synchronous, asynchronous, and active messages (used for event driven applications). The library also includes functions for global operations (reductions, scans, broadcasts, barriers) and parallel I/O. CMMD has no support for process control or virtual machine control.

Many other message passing systems provide similar functionality to these three. PVM, MPI, and CMMD are of particular interest to us since they are the "supported" message passing systems for the T3D and the CM-5.

3 The Need for Performance

Our software efforts are targeted towards high performance software for MPPs and SMPs (Symmetric Multi-Processors). Our focus is not on harnessing the latent power of desktop workstations. Nor is it in running a single program on several supercomputers. Given this, several key differences should be noted between PVM, MPI, and CMMD.

PVM is widely available for most Unix workstations and for many common supercomputers and MPPs. Portability, through support of heterogeneous data types and computers, is a main goal. PVM's main weakness is that it is not high performance. One example of this is that past versions utilize a daemon process on each computer node which was involved in communications. Recent versions of PVM allow these daemons to be optionally by-passed; however, performance is still lacking as will be shown.

MPI is a recent message passing system and is not widely available at this time. MPI includes numerous primitives (far more than PVM), except for process management. While efficiency is a main goal for MPI, our benchmarks on the T3D show that it is lacking as well. MPI, like PVM, has the goal of supporting heterogeneous data types and computers.

CMMD differs from PVM and MPI in that it is not available on anything other than the CM-5; however, it is does have a large user base since it was the only supported message passing system available on the CM-5 until recently (PVM was recently ported to the CM-5 by TMC). CMMD has a small set of primitives which are efficient, simple, but complete. It has the basic communications primitives, active messages and the most commonly used global operations.

CMMD was designed for interprocessor communications within the CM-5 and not with processes external to the MPP. This allows for several optimizations. Since the library is not designed to communicate with heterogeneous processors or data types, it avoids unnecessary data conversion and a plethora of different primitives for various data types. CMMD also takes advantage of the underlying hardware support. For

example, it utilizes both the data network and the control network in the CM-5. In particular, the control network is used in global communications operations such as reductions and broadcasts.

ACLMPL was developed with similar constraints and goals as CMMD: message passing within a single multiprocessor machine (MPPs and SMPs), efficient global operations, and sufficient primitives without trying to be all encompassing. As will be shown, this results in a message passing system, for both synchronous and asynchronous communications primitives, that is faster than PVM and MPI.

4 Implementation

ACLMPL is split into two groups: the synchronous communications primitives and the asynchronous primitives. On top of the synchronous primitives are layered the global communications primitives. Splitting synchronous and asynchronous primitives into two separate groups, with no overlap, allows for greater optimization than would be possible otherwise. For example, layering synchronous on top of asynchronous, is possible but it introduces additional overhead (extra function calls, buffering, etc.). Additionally, the timings will show that synchronous communication can be faster than asynchronous communications.

The following sections will describe the implementation of ACLMPL on the T3D. Later sections will briefly discuss the CM-5 and SGI implementations.

4.1 T3D Synchronous Communications

The synchronous message passing API in ACLMPL was implemented first. Synchronous message passing has some potential performance advantages over asynchronous methods since there is no need for intermediate buffering. Data can be sent directly from the sender to the receiver with no need for additional data copying. This can result in much higher bandwidth and lower latency than is possible with an asynchronous protocol. The tradeoff is that computation cannot be overlapped with communications[2].

A simple protocol built on the CRI SHMEM library `shmem_put()` function, which is faster than `shmem_get()`, is used as the lowest level communications primitive on the T3D [1]. Figure 1 shows the protocol used to send data between two processes on two separate Processing Elements (PEs). The receiving PE first writes a request block to the sending PE which contains the receive buffer address, its buffer length, and a control flag. The request block totals 16 bytes. Each PE has an array of request blocks, indexed by receiving PE. This avoids the need for locks on the request blocks since each block has only one writer.

The sending PE blocks, via a spin-wait loop checking the control flag, until this request block arrives. Once the request block is received by the sender, the sender initiates a `shmem_put()` from the local send buffer address to the receiver's buffer address which is taken from the receive request block. Finally, after the data is transferred, a completion block is transmitted back to the receiver, indicating the size of the

[2]Except through the use of a thread package which allows multiple threads of execution on each PE.

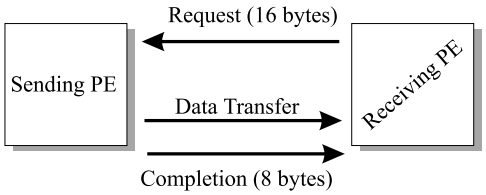

Figure 1: Synchronous Protocol

transfer, in bytes, and a flag value (DONE) indicating the transfer has completed. This completion block consists of 8 bytes.

The receiver, after initiating the request, waits in a spin-wait loop for its flag to change to DONE. Once the flag changes to DONE, both sender and receiver return. The synchronous protocol requires one round trip between the sending and receiving PEs and a total of 24 bytes of overhead information. This results in very low end-to-end latency (4.5 microseconds for a one word message transmitted between direct neighbor PEs) and high bandwidth (greater than 100MB/sec for one-to-all and all-to-one communication patterns).

Based upon the synchronous protocol there are three user callable functions: `send()`, `receive()`, and `send_and_receive()` (send to one PE and receive from another PE, possibly the same).

4.2 T3D Asynchronous Protocol

Efficient asynchronous message passing exposes a number of implementation challenges on the T3D. Unlike the synchronous case, the asynchronous algorithm must address buffer management, race conditions, and synchronization issues. Additionally, at least one extra data copy will be necessary between the application memory and a buffer within the message passing library, which is avoided in the synchronous case. Since word aligned `memcpy()` speeds on the T3D are only 170MB/sec (approximately), it is important to minimize the number of data copies in order to achieve high bandwidth.

Our approach to the buffer management problem follows that used in the Illinois Fast Messaging library [7]. As in FM, we use the fetch-and-increment registers on the T3D to allocate remote buffers from a fixed sized pool of buffers as shown in Figure 2. A sending PE reads the fetch-and-increment register on the receiving PE. The read operation returns the current value of the fetch-and-increment register, while atomically incrementing it as well. If the fetch-and-increment register is out of the bounds for the buffer pool, the sender must block until the receiver removes messages from the buffer pool and resets the fetch-and-increment register. If it is in bounds, the value read gives an index into the receiver's buffer pool, providing a buffer which the sender has exclusive access to. The sender transfers the message data to this buffer, via `shmem_put()`, and transfers a flag value DONE, indicating the transfer is complete.

The receiving PE first checks a linked list of sent-but-not-yet-received messages for a message that matches the receive request. If a matching message is found, the data is `memcpy`'d to the caller's buffer and the linked list node is freed. If a matching message is not found in the linked list, the buffer pool itself is

scanned for a matching message. If a matching message is found, the data is `memcpy`'d to the caller's buffer and the buffer pool slot is marked as RECEIVED. In most cases, the linked list is empty and a matching message is found directly from the buffer pool, resulting in a one data copy, in addition to the `shmem_put()`.

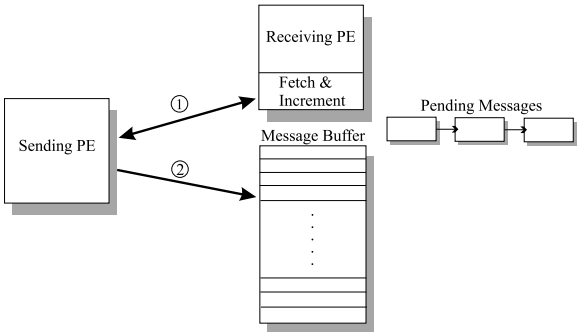

Figure 2: Asynchronous Protocol

Each PE periodically checks whether its fetch-and-increment register has overflowed. This check is made each time a send or receive request is processed. The check can be accomplished by examining the last buffer in the buffer pool to see if it is marked as DONE or RECEIVED. If the fetch-and-increment register is out of bounds, all messages in the buffer pool are copied out into a linked list of sent-but-not-yet-received messages, and the fetch and increment register is reset to zero. This allows blocked senders to resume.

The user callable functions for the asynchronous protocol are asynchronous send, asynchronous receive, and blocking asynchronous receive. The two receives differ in that the first returns immediately if a message is not available. The other will block until a message has been received.

4.3 Global Operations

The global operations consist of a broadcast and a reduce primitive. The reduce primitive is extensible in that the user can write a reduction operator.

Broadcast and reduce global operations are implemented in ACLMPL using efficient tree based algorithms [2]. For simplicity, both broadcast and reduce use PE 0 as the root processor, though the algorithms can be generalized to handle any root PE.

A broadcast from PE 0 is sent in $\log(P)$ phases, where P is the partition size. In the first phase, only PE 0 is active and the broadcast is sent from PE 0 to PE $\frac{P}{2}$. In the second phase, PE 0 and PE $\frac{P}{2}$ are active and each sends to PE $self + \frac{P}{4}$. In the i^{th} phase, PEs which have received the data forward the data onto the PE whose number differs only in the $(log(P) - i)^{th}$ bit. This is a well known algorithm whose complexity is $\mathcal{O}(NlogP)$, where N is the size of the broadcast and P is the partition size[3].

[3]Technically, this time bound and those that follow assume a hypercube interconnection network, though empirical evidence indicates that they match well to measured performance on the T3D's 3D torus network as well.

The reduction operation can use the same tree structure used in the broadcast but in reverse, again yielding a $\mathcal{O}(NlogP)$ time bound. Initially all PEs are active. In the i^{th} phase of the algorithm, the PEs which have a 1 in the i^{th} bit of their PE number send to the PE whose PE number is identical except for a 0 in the i^{th} bit. The sending node becomes inactive, while the receiving node combines the received data with its own and proceeds to the next phase. At the end of the reduction, PE 0 holds the entire reduced array.

Note that in each phase of the reduction, as we move up to the root of the tree, fewer PEs are participating in the operation. This suggests that a more efficient algorithm could be devised which utilizes all the PEs during every phase. We first made this observation in a special case of the reduction algorithm: image compositing in a sort last volume renderer [8]. In our *binary-swap* reduction algorithm we split the array being reduced in half at each phase of the algorithm and keep all PEs active throughout all phases.

In the i^{th} phase of the algorithm, two PEs whose PE numbers differ only in the i^{th} bit split their reduction array into two sub-arrays of equal size. One PE takes the lower sub-array while the other takes the upper sub-array. The two PEs exchange data, combine the received data with their own, and both proceed to the next phase. At the end of the final phase, the entire array has been reduced, but it is distributed across all the PEs. A final gather stage brings the result together in PE 0. The binary swap reduction algorithm runs in $\mathcal{O}(N)$ time when the array size N is much larger than the partition size P. On the T3D we have found that $N \geq 1024$ is sufficient for binary swap reduction to outperform the simple tree based algorithm.

As previously mentioned, the global operations are built upon the synchronous primitives. Since all PEs must participate in a global operation, asynchrony is not needed. Furthermore, the synchronous primitives are faster since they do not do any buffering of data.

4.4 ACLMPL for the CM-5

Since ACLMPL closely mimics CMMD, the CM-5 version consists mainly of source to source transformations. This results in no overhead for using ACLMPL on the CM-5. The only ACLMPL functions implemented on the CM-5 are the reduce and broadcast primitives. This allows the user to write his or her own reduction operations, which is not supported by CMMD. Additionally, we have found our version of broadcast to be faster than the CMMD version for larger message sizes (approximately 2K bytes).

4.5 ACLMPL for the SGI

The Silicon Graphics version of ACLMPL is based upon IRIX specific interprocess communication (IPC) functions[4]. These functions allow for the creation and management of a shared pool of memory which is used to facilitate the communication of messages between processors. In addition, these routines support barriers, semaphores, and locks.

[4]The IRIX routines have better performance than the standard AT&T System V Release V IPC routines. See the SGI Insight manual, *Topics in IRIX Programming* for more details.

4.5.1 Synchronous Protocol

Once the shared pool of memory, known as a shared arena, is created, a portion of it is set aside for internal data structures. These data structures allow processors to share both synchronization and message information. This closely resembles the synchronous implementation on the T3D.

The first step in sending a message is to allocate a region in shared memory large enough to hold the incoming message. Next, the sender copies the message into this region and waits for the receiver to arrive. Once the receiver has arrived, the sender provides the size and location of the allocated block in shared memory.

The receiver's first step is to alert the sending processor that it is ready and waiting for the message. It then waits for the size and memory location of the message from the sender. Once the sender has provided this information, the message is copied out of the shared arena and into the receiver's address space. The receiver's final task is to free the shared memory buffer and inform the sender the message has been received.

A clear disadvantage of this approach is the memory copies to and from the shared arena. However, this may be overcome by using the direct memory mapping routines supplied by the IRIX operating system.

4.5.2 Asynchronous Protocol

The asynchronous message passing routines create a second shared arena – this allows for a clean separation from the synchronous implementation. However, if memory is scarce, both protocols may share the same arena. The approach used by the SGI asynchronous protocol also closely matches the T3D implementation described above.

During initialization, a data structure is created in the shared arena where both incoming messages and their status will be stored. This region of memory is of a fixed size and the sending processors are responsible for copying messages into a free location if one exists. Since the incoming message buffer is of a fixed size, the sending processors are blocked until a free slot becomes available.

The current implementation lacks several optimizations, such as using direct memory mapping, which can increase performance. Future development will address this optimization and others.

In addition, ACLMPL has been implemented on top of PVM's `psend()` and `precv()` functions. This not only provides us with a more portable version of the library, but can also help in the early stages of application development and debugging without the use of an MPP.

5 Timings

Numerous benchmarks were performed on ACLMPL, MPI[5], PVM, and SHMEM using the T3D. For each of these packages we attempted to write the most efficient programs possible. For example, in PVM we used `psend()` and `precv()` instead of using the other routines which pack and unpack the data. Six different test cases were run on various partition sizes and for various message sizes. The six cases are: one PE communicating with all others (one-to-all), all PEs communicating with all others (all-to-all), all PEs communicating with one PE (all-to-one), global reduction, global broadcast, and latency. Performance figures are included for SHMEM, in addition to the message passing systems, to give a reference for how they compare to using shared memory for communications.

The six cases were chosen for the following reasons. One-to-all is typical of initial data distribution, such as when one PE is responsible for reading a file and distributing parts of it to different PEs. Similarly, all-to-one is representative of gathering results back from all PEs for performing serial I/O. All-to-all is indicative of worse case, general communications. Global reduction and broadcast are included since they are very common global operations. The latency benchmark measures the overhead involved in sending very short messages (1 word) and measures the minimum overhead in sending short messages. Because many of the graphs exhibit similar curves, we have chosen a representative few for this paper[6].

Figure 3 and Figure 4 show the performance curves for the all-to-all case on 2 and 128 PEs. The Y axis shows throughput and the X axis shows message size in bytes. Several interesting features can be seen.

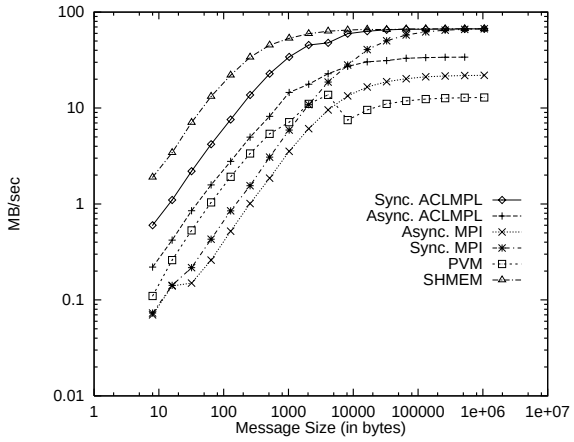

Figure 3: All-to-all communications on 2 processors.

Throughput for all of the message passing systems increases greatly until the message size becomes sufficiently large (greater than 1K bytes) and then tapers off. Synchronous ACLMPL is as fast as all of the other message passing systems for all cases. Additionally, for partitions greater than 2 PEs and for message

[5]The T3D MPI implementation was from EPCC. The MPICH implementation could not properly execute the test programs.

[6]All performance results are available via URL, `http://www.acl.lanl.gov/Viz/aclmpl_timings.ps`

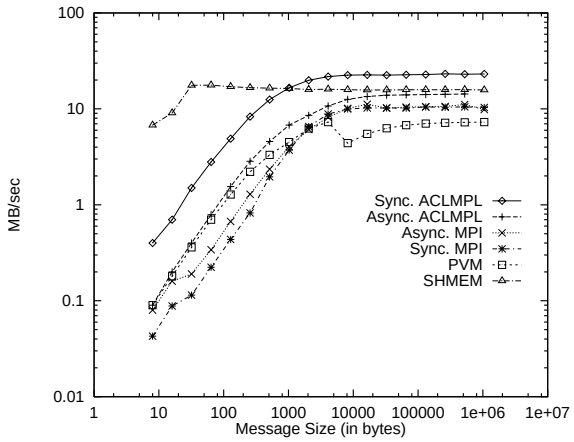

Figure 4: All-to-all communications on 128 processors.

sizes greater than 1K bytes, it is faster than either shared memory or the other message passing systems. This seems curious at first since ACLMPL is built on top of SHMEM. The explanation is that the SHMEM version floods the T3D network and causes collisions, thus reducing performance. Synchronous ACLMPL requires serialization (a PE can only receive from one sender at a time) which helps avoid saturating the network switches, thus resulting in greater performance.

As the partition size increases, maximum throughput for the all-to-all case decreases from 67 MB/s to 23 MB/s. The kink in the PVM curve is due to a different, internal algorithm used by PVM for handling large messages[7]. Finally, asynchronous ACLMPL functions are also faster than the other message passing systems for partitions containing 32 or more PEs. For partitions smaller than 32 PEs, ACLMPL is faster for message sizes less than 8K bytes.

Figure 5 shows performance curves for all PEs sending to one PE on a 128 PE partition. The synchronous version of ACLMPL is faster than the other message passing systems, as is the asynchronous version for messages less than 8K bytes. SHMEM is faster than ACLMPL in all cases since there is not the abundance of collisions on the network as there is with the all-to-all case. Maximum throughput is greater than 110 MB/s for synchronous ACLMPL.

Curiously, the spike in the PVM curve in the one-to-all case changes direction from all other test cases. Unfortunately, we have not been able to explain the direction change in the spike without access to the PVM source code for the T3D.

The one-to-all case, Figure 5, exhibits similar performance curves with the exception that PVM seems to do better than it did in the all-to-one case.

Figure 7 and Figure 8 show broadcast times for 2 and 128 PEs Both graphs exhibit similar curves with the exception of the PVM curve. As the number of PEs increases, the upward spike in the PVM curve grows. It should also be noted, that as the number of PEs increases, the time for all message passing systems

[7]See the Cray T3D PVM documentation on the `PVM_DATA_MAX` environment variable

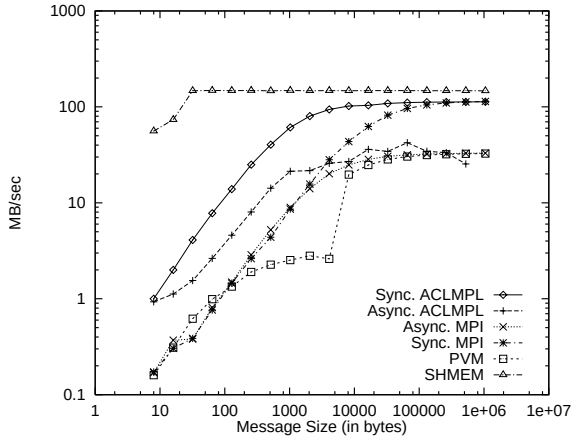

Figure 5: All-to-one communications on 128 processors.

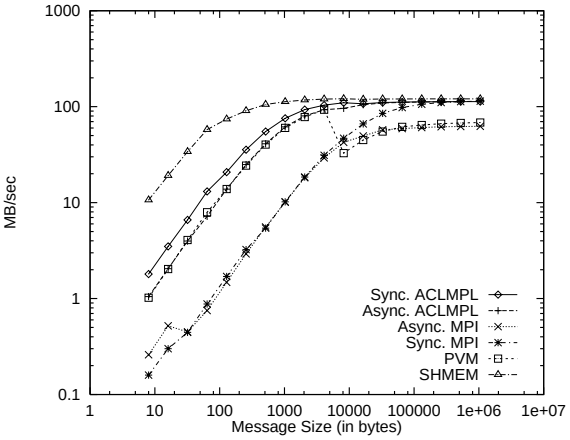

Figure 6: One-to-all communications on 128 processors.

increases regardless of message size.

Figure 9 shows times to perform a global reduce using 128 PEs. MPI is significantly slower than ACLMPL; and PVM performs well for small messages but then degrades for larger messages.

Table 1 shows the latency times for sending a one word message. Both the MPI synchronous and asynchronous versions incur significant overhead in sending a short message (greater than 8 times that of ACLMPL synchronous messages). It should be noted that the T3D is extremely instruction cache sensitive and that cache coherency and alignment will greatly affect these timings.

Table 2 presents performance numbers for 1024 byte messages on a 32 PE partition which seems to be a commonly used size. The numbers for all-to-all, all-to-one, and one-to-all are in megabytes per second; and the numbers for broadcast and reduce are in seconds. For the first three cases, the synchronous functions in ACLMPL are approximately between 4 and 7 times faster than the other message passing systems, and

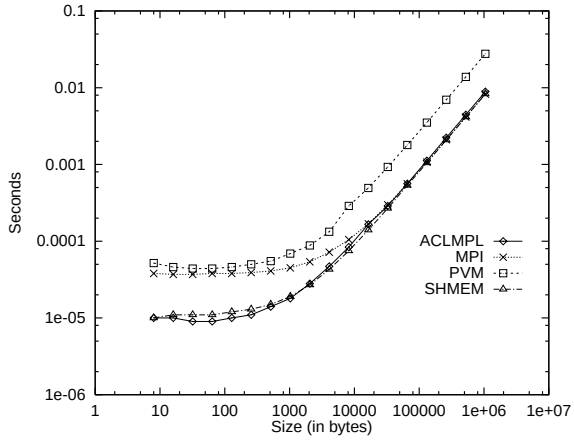

Figure 7: Broadcast on 2 processors.

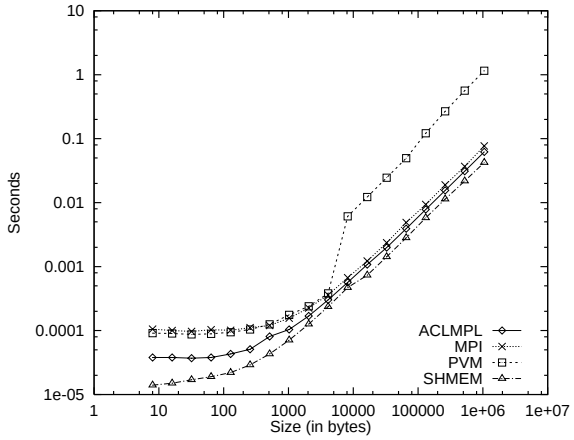

Figure 8: Broadcast on 128 processors.

broadcast and reduce are roughly 10 to 80 percent faster.

6 Results

While ACLMPL recently grew out of the efforts of the visualization group at the Advanced Computing Lab, it is a general purpose communications library. One example of its use is in a molecular dynamics application for massively parallel computers. The application is used to simulate molecules containing several hundreds of millions of atoms. In 1993 it won a Gordon Bell prize for performance (it was able to sustain >53 Gflops on a 1024 node CM-5). It should be noted that at that time the application was based on CMMD. It is currently being ported to ACLMPL.

ACLMPL has been used in two newly developed visualization applications. One is a sphere renderer that

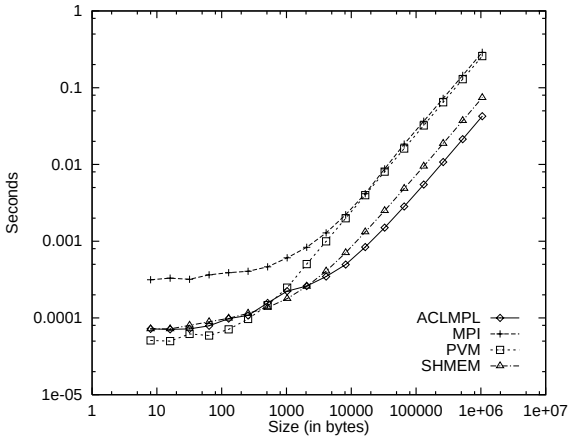

Figure 9: Reduce on 128 processors.

Protocol	Time (μ seconds)
ACLMPL (sync)	5
ACLMPL (async)	15
PVM	25
MPI (sync)	47
MPI (async)	40

Table 1: Latency

is used by the molecular dynamics project for displaying their data. The renderer can be used as either a stand-alone program or as a MIMD callable library. As a stand-alone program, the renderer can be used either interactively with an X11 graphical user interface (GUI) or in a batch mode. Images are either displayed in an X11 window, on a HIPPI frame buffer, or written to disk. Rendering rates on the T3D are approximately 660K spheres per second. For comparison, a SGI Onyx graphics workstation can sustain roughly 19K spheres per second.

The second visualization application is a renderer for volumetric data based upon Binary-Swap Compositing [6]. The renderer distributes a 3D data set to the PEs. Each PE is responsible for rendering its own subvolume. After each PE is done, the subimages are composited together using binary-swap. The user can interact with the renderer either through an X11 interface or through AVS. The renderer can generate approximately 4 frames per second using 128 PEs to render a 128^3 data set into a 256 x 256 image that is displayed on a HIPPI frame buffer.

	ACLMPL (sync)	ACLMPL (async)	MPI (sync)	MPI (async)	PVM
alltoall	19.00	7.93	4.71	4.58	4.43
alltoone	61.90	36.43	10.70	10.72	8.94
onetoall	74.00	60.61	10.70	9.96	57.02
bcast	0.000076	–	0.000135	–	0.000138
reduce	0.000162	–	0.000452	–	0.000180

Table 2: Performance for 1KB messages on 32 PE's.

7 Conclusions

ACLMPL was developed with two goals in mind: to provide high throughput, low latency communications for message passing applications, and to provide portability. As previously shown, ACLMPL is approximately 4 to 7 times faster than either MPI or PVM on the Cray T3D for general communications and 10 to 80 percent faster for global communications. This is significant to MPP applications since slow communications will reduce performance and scalability.

Since ACLMPL is based very closely on TMC's CMMD, we can preserve our software investment. Additionally, we have found ACLMPL to be quite portable to other platforms while still retaining efficiency.

While we don't expect, nor want, ACLMPL to become the "new message passing standard", we would hope that it can be seen as a challenge to those who implement message passing systems. ACLMPL should be viewed as proof that it is possible to develop a portable, usable, high performance message passing system for MPPs.

Finally, four major points should be noted. First, synchronous message passing is inherently simpler than asynchronous message passing. This is because buffer management and additional data movement can be avoided. These optimizations should be used. Second, efficient global communications algorithms exist and should be used; otherwise, scalability to large partition sizes is impaired. Third, on the T3D efficient buffer management can be performed using the fetch-and-increment facilities. Last, while portability is a highly desirable trait, perhaps performance should be equally important when supplying message passing systems for use within a MPP. MPI tends more towards this balance than does PVM, although additional performance gains should still be possible as we have demonstrated.

References

[1] Ray Arriuso and Allan Knies. SHMEM User's Guide. *Cray Technical Report*, May 1995.

[2] M. Barnet, P. Littlefield, D.G. Payne, and R. van de Geijn. On the Efficiency of Global Combine Algorithms for 2-D Meshes With Wormhole Routing. *Journal of Parallel and Distributed Computing*,

24, 1995.

[3] Thinking Machines Corporation. CMMD User's Guide. *TMC Reference Manuals*, 1993.

[4] Al Geist et. al. PVM 3 User's Guide and Reference Manual. *Oak Ridge National Laboratory*, September 1994.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applicationsi*, 8(3 & 4), 1994.

[6] Charles D. Hansen, Michael F. Krogh, James S. Painter, Guillaume Colin de Verdière, and Roy Troutman. Binary swap volumetric rendering on the t3d. *Cray Users Group Conference, Denver, CO*, March 1995.

[7] Viay Karamcheti and Andrew A. Chien. A Comparison of Architectural Support for Messaging on the TMC CM-5 and the Cray T3D. *To appear in the Procceedings of ISCA '95, Santa Margherita, Italy*, June 1995.

[8] Kwan-Lui Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary swap compositing. *IEEE Computer Graphics and Applications*, 14(4), July 1994.